

AD-A258 999



AFIT/GCE/ENG/92D-01


DTIC
ELECTE
JAN 11 1993
S C D

PARALLEL SIMULATION OF
STRUCTURAL VHDL CIRCUITS ON
INTEL HYPERCUBES

THESIS

Thomas A. Breeden
Captain, USAF

AFIT/GCE/ENG/92D-01

93-00085


Approved for public release; distribution unlimited

93 1 4 055

PARALLEL SIMULATION OF
STRUCTURAL VHDL CIRCUITS ON
INTEL HYPERCUBES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

DTIC QUALITY INSPECTED 8

Thomas A. Breeden, B.S.E.E
Captain, USAF

Dec, 1992

Approved for public release; distribution unlimited

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Acknowledgements

I'd like to thank my thesis committee for their enthusiastic support and guidance. Maj Kanzaki kept my nose to the schedule, and Maj Christensen was always available when I needed to iron out an issue or bounce an idea off of a "greater mind." Dr. Hartrum was especially helpful in every area of this effort. He has an uncanny ability to create a workable model of *anything* on his chalkboard. After a session of "chalk-talk," I could always go back to the computer and implement our design in a straightforward manner. Even more uncanny is Dr. Hartrum's method of treating you like an equal and making you feel like what you're doing is truly worthwhile—a highly effective management technique that I hope to take with me.

This research could not have been accomplished without a significant amount of technical support from the following people:

- Ron Comeau demonstrated that extracting intermediate C code from Intermetrics' VHDL compiler was an effective method for generating parallel VHDL simulations. He is responsible for identifying the methods for transforming the C code, the key data structures, and the basic sequential simulation algorithm. From there, I went my own way; however, my "successes" could not have been achieved without his initial investigation and design. Also, the VHDL source code for the adders were taken from Comeau's research.
- Dave Daniel provided the VHDL source code for the shifter.
- Maj Kanzaki created the VHDL wallace tree multiplier, which is the largest structural circuit simulated.
- Scott VanHorn brought me up to speed on SPECTRUM in less than a week. He was also a great help in designing that very tricky receive filter.

- Maj Christensen has begun work on a “VHDL graph tool,” which will be used in the future to generate and test circuit partitioning strategies. I used this tool to generate the uniform random distributions of behaviors to logical processes for the shifter and multiplier.
- Rick Norris provided a tool to automatically generate `lp.arcs` files, which SPECTRUM requires. I found generating these files “by hand” to be the greatest source of user error in running the parallel simulations.

To my best friend and brother, Tim, thanks for keeping me laughing with the e-mail. Good luck at Carnegie Mellon.

Finally, to Barbara and the boys, I present this modest thesis as evidence that I was indeed at school all those nights. Thanks for your patience.

Thomas A. Breeden

Table of Contents

	Page
Acknowledgements	ii
Table of Contents	iv
List of Figures	ix
List of Tables	xii
Abstract	xiii
 I. Introduction	 1
1.1 Background.	1
1.2 Problem Statement.	2
1.3 Research Objectives.	2
1.4 Assumptions.	3
1.5 Scope.	4
1.6 Limitations.	4
1.6.1 VHDL Source Code Limitations for VSIM.	4
1.6.2 Postprocessor Limitations.	5
1.6.3 VSIM limitations.	6
1.7 Thesis Overview.	6
1.8 Summary.	7
 II. Background	 8
2.1 Overview.	8
2.2 Traditional Simulation.	8
2.3 Distributed Simulation.	9
2.3.1 General Performance Model.	10

	Page
2.3.2 Speed-Up and Efficiency of Distributed Simulations.	10
2.3.3 Distributed Simulation Protocols.	10
2.4 Overview of SPECTRUM.	14
2.5 Other Parallel VHDL Research.	16
2.6 Summary.	18
III. Methodology	19
3.1 Introduction.	19
3.2 Overview.	19
3.3 Data Structures.	21
3.4 Sequential Simulation Cycle.	23
3.5 Active List Management.	27
3.5.1 Transport Delays.	27
3.5.2 Inertial Delays.	27
3.6 Transformation of Intermediate C Code.	29
3.7 Parallel VHDL Simulation.	30
3.7.1 SPECTRUM and VSIM.	30
3.7.2 The SPECTRUM/VSIM Filters.	31
3.7.3 Modifications to VSIM for Parallel Simulation.	33
3.8 Summary.	38
IV. Implementation	39
4.1 Introduction.	39
4.2 Postprocessor Implementation.	39
4.2.1 Transformation Steps.	43
4.2.2 Lex Descriptions of the Transformation Steps.	47
4.3 Interfacing VSIM with SPECTRUM.	49
4.3.1 Main SPECTRUM Functions.	49

	Page
4.3.2 Implementation of SPECTRUM Filters for VSIM.	52
4.3.3 Termination.	53
V. Results	54
5.1 Introduction.	54
5.2 Program Validation.	54
5.3 Circuit Partitioning.	57
5.4 Explanation of Charts.	58
5.5 Circuit Simulations.	58
5.5.1 Carry Save Adder.	58
5.5.2 Carry Propagate Adder.	60
5.5.3 Carry Lookahead Adder.	63
5.5.4 Shifter.	70
5.5.5 Multiplier.	70
5.6 Performance vs. Test Vector Quantity.	70
5.7 Multitasking LPs on one Physical Processor.	77
5.8 Performance with Output Enabled.	80
VI. Conclusions/Recommendations.	81
6.1 Research Summary.	81
6.2 Conclusions.	81
6.3 Recommendations for Further Research.	82
6.3.1 Parallel Simulation Recommendations.	82
6.3.2 Improving the Postprocessor.	83
6.3.3 Expanding the VHDL subset.	83
6.3.4 Other Recommendations.	84
Appendix A. Definitions	86
A.1 Discrete-Event Digital Simulation Definitions.	86
A.2 VHDL Definitions.	86

	Page
Appendix B. AFIT Parallel VHDL User's Guide	90
B.1 Overview.	90
B.1.1 Introduction.	90
B.1.2 Process.	90
B.1.3 Related Files.	90
B.2 Implementation.	93
B.2.1 Introduction.	93
B.2.2 Generating VHDL Source Code.	93
B.2.3 Setting up a User Library for Circuit Models.	93
B.2.4 Compiling, Model Generating, and Building.	94
B.2.5 Extracting and Transforming Intermediate C Code.	94
B.2.6 Running VSIM on a Sequential Machine.	97
B.2.7 Generating Partitioning Strategies.	98
B.2.8 Running VSIM on a Parallel Machine.	99
B.3 Example: An Edge-Triggered D Flip-Flop.	99
B.3.1 Introduction.	99
B.3.2 VHDL Source Code.	100
B.3.3 Compiling, Model Generating, Building, and Simulating under In- termetrics.	100
B.3.4 Using the Postprocessor to Generate Intermediate C Code. . . .	102
B.3.5 Sequential Simulation with VSIM.	103
B.3.6 Extracting Behavior Information using VMAP.	108
B.3.7 Generating .arcs and .map Files for Partitioning.	110
B.3.8 Parallel Simulation.	112
B.3.9 Summary.	113

	Page
Appendix C. Subset of VHDL Source Code for Parallel Simulation	122
C.1 Logic Gates.	122
C.2 Structural Connection of Logic Gates.	123
C.3 Test Bench and Input Vectors.	125
C.4 Configuration Descriptions.	127
Appendix D. Design of the Wallace Tree Multiplier	131
Appendix E. Summary of Performance Data	139
Appendix F. New Postprocessor Steps	142
Appendix G. Key Source Code	144
G.1 vspec_init().	144
G.2 startup().	145
G.3 send_signal().	145
G.4 receive_signal().	146
G.5 null_post_ftr().	147
G.6 able_to_proceed()	147
G.7 safetime().	148
G.8 send_nulls().	148
G.9 null_get_ftr().	149
Bibliography	151
Vita	153

List of Figures

Figure	Page
1. Response Measurement from a Discrete-Event Simulator (27)	9
2. A Distributed System That Does Not Progress (25:56)	12
3. A Distributed System That Deadlocks (25:56)	12
4. Block Diagram of the SPECTRUM Testbed (31)	15
5. Simulation Session Using Intermetrics' Toolset (10:3-8)	20
6. Parallel Simulation Session	21
7. Basic Structure for Behavior Instances	22
8. Basic Structure for Signal Records	22
9. Behavior List Structure	23
10. Active Record Structure	23
11. Interrelationship of VHDL Simulation Data Structures (10:3-14)	24
12. The VHDL Simulation Cycle (10:3-15)	2

Figure	Page
26. Result of Reading Compilation Script by pbuild	42
27. Relationships of the Postprocessor Files	43
28. Regular Expressions Required to Identify Data to be Transformed	47
29. Function Calls and Actions Defined for Each Regular Expression	48
30. Example Postprocessor Report	50
31. Sample Intermetrics Output for Carry Lookahead Adder	55
32. Sample VSIM Output for Carry Lookahead Adder	56
33. Schematic Diagram of the 8-bit Carry Save Adder (10)	59
34. Performance of the Carry Save Adder on the iPSC/2	61
35. Performance of the Carry Save Adder on the iPSC/860	62
36. Schematic Diagram of the 8-bit Carry Propagate Adder (10)	63
37. Performance of the Carry Propagate Adder on the iPSC/2	64
38. Performance of the Carry Propagate Adder on the iPSC/860	65
39. Schematic Diagram of the 8-bit Carry Lookahead Adder (10)	66
40. Four-LP Partition of the Carry Lookahead Adder (Lower	

Figure	Page
53. Overview of Parallel Simulation Session	91
54. Section of .cshrc File for Setting up Intermetrics VHDL in the AFIT VLSI Lab . . .	93
55. Example Initialization of Intermetrics VHDL	94
56. Example Format for One LP in an lpx.arcs File	98
✓57. Edge-Triggered D Flip-flop	114✓
58. VHDL Descriptions of Two- and Three-Input NAND Gates	115
59. Structural VHDL Description of Edge-triggered D Flip-flop	116
60. VHDL Description of Test Bench for Edge-triggered D Flip-flop	117
61. Schematic of Test Bench for Edge-triggered D Flip-flop	118
62. VHDL Description of Configuration File for Edge-triggered D Flip-flop	119
63. VHDL Report Description for Edge-triggered D Flip-Flop	119
64. Shell Script for Compiling, Model Generating, Building, and Simulating the Edge-triggered D Flip-flop using Intermetrics' Simulator	120
65. Edge-Triggered D Flip-flop Labeled with Behavior Id Numbers	120
66. Edge-Triggered D Flip-flop Partition	

List of Tables

Table	Page
1. Length of Intermediate C Code Circuit Descriptions	39
2. Files Necessary for Maintenance and Operation of the Postprocessor	91
3. Files Necessary for Maintenance and Operation of VSIM	91
4. Files Necessary for Maintenance and Operation of Parallel VHDL Simulations using SPECTRUM	92
5. Files Necessary for Maintenance and Operation of VMAP	92
6. Other Files	92
7. Example Format for the lpx.map File	99
8. Results of 1, 2, and 3 LP Configurations for the Edge-triggered D Flip-flop	114
9. Summary of Performance Data	140
10. Summary of Performance Data (cont.)	141

Abstract

Many VLSI circuit designs are too large to be simulated with VHDL in a reasonable amount of time. One approach to reducing the simulation time is to distribute the simulation over several processors. This research creates an environment for designing and simulating structural VHDL circuits on the Intel iPSC/2 and iPSC/860 Hypercubes. Logic gates and system behaviors are partitioned among the processors, and signal changes are shared via event messages. Circuit simulations are run over the SPECTRUM parallel simulation testbed, and the null-message paradigm is used to avoid deadlock. Structural circuits ranging from forty to over one thousand logic gates are correctly simulated. Although no attempt is made to find *optimal* partitioning strategies, speedups are obtained for some configurations.

PARALLEL SIMULATION OF STRUCTURAL VHDL CIRCUITS ON INTEL HYPERCUBES

I. Introduction

1.1 Background.

Advances in Very Large Scale Integrated (VLSI) circuit technology increase the transistor count on a chip by about 25% per year, doubling every three years (17:17). In order to efficiently design increasingly complex VLSI circuits, designers use simulation tools to validate their circuits prior to fabrication. In 1979, the Department of Defense (DOD) started the Very High Speed Integrated Circuit (VHSIC) program to employ the use of high density VLSI circuits in military systems. The VHSIC Hardware Description Language (VHDL) program began in 1983 to standardize the tools needed to efficiently design and test these circuits (13, 22).

Many circuit designs are too complex to be simulated with VHDL in a reasonable amount of time. In an effort to improve VHDL's performance, the Defense Advanced Research Projects Agency (DARPA) has sponsored the QUEST project, whose goal is a thousand-fold speed-up in VHDL simulation (28:1-1). One approach to reducing the simulation time is to distribute the simulation of the design over several processors. If VHDL's capabilities could be effectively mapped to a parallel processor, the simulation would be faster and users could design and run more complex circuits. Efforts at AFIT have centered on creating a parallel implementation of VHDL for this purpose.

In 1991 AFIT investigated the data structures of Intermetrics' sequential VHDL simulator and demonstrated a way to intercept intermediate C code from Intermetrics' compiler, transform it,

and run parallel simulations on the Intel iPSC/2 Hypercube (10). This research effort composes the tools necessary to create and run structural VHDL simulations on the Intel iPSC/2 and iPSC/860 Hypercubes.

1.2 Problem Statement.

AFIT has investigated implementing a parallel VHDL simulator to decrease the simulation times of VLSI circuits; however, an automated method for creating parallel VHDL circuit descriptions, a correct parallel simulator, and a common distributed testbed are necessary to generate and simulate large VHDL circuit models.

1.3 Research Objectives.

The main objective of this thesis is to demonstrate and test the capability of mapping large sequential VHDL circuit descriptions to distributed processing systems. The main goals are to

- automate the procedures for generating hierarchical, structural VHDL models.
- create a VHDL simulator that correctly simulates structural VHDL circuit descriptions and is flexible enough to partition simulations among the processors of a distributed system.
- provide a common testbed to facilitate experimentation with parallel simulation protocols and investigation into optimizing circuit partitioning strategies.
- demonstrate the simulator with several VHDL models.
- determine if speedup can be achieved through the use of parallel simulations.

1.4 Assumptions.

Comeau did the preliminary research into transforming Intermetrics' VHDL models into models that can be simulated in a parallel environment. The following assumptions build upon Comeau's research (10:1-3):

- While strict meanings of "parallel" and "distributed" processing systems vary from source to source, AFIT has generally accepted "parallel processing" to indicate processing on a single computer composed of multiple processors, while "distributed processing" refers to processing among several "independent" computers across a network. Nonetheless, as with Comeau's thesis, this research uses the terms "parallel" and "distributed" interchangeably throughout.
- The parallel computers used for development and research are the Intel iPSC/2 and iPSC/860 Hypercubes.
- Source code is written in the standard C programming language (non-ANSI).
- To further research efforts for both DARPA and AFIT and stay consistent with the AFIT environment, the Chandy-Misra conservative synchronization algorithm for event-driven simulations is used. In this thesis, the null-message protocol is implemented via the use of a parallel simulation environment known as SPECTRUM (Simulation Protocol Evaluation on a Current Testbed using Reusable Modules) (32).
- The output from the analyze, model generate, and build phases of the Intermetrics VHDL compiler are correct and accessible.
- The VHDL test cases are within the VHDL subset that is used to demonstrate parallelized VHDL.
- VHDL source code is compiled and model generated in Intermetrics VHDL, Version 2.1, September 1990.

1.5 Scope.

Comeau outlined ten steps to transform Intermetrics' intermediate C code into modules that can run on a parallel VHDL simulator (10:4-6). These operations are automated, and new steps are added to reduce unnecessary function calls and enhance simulator capabilities.

A new parallel simulator, VSIM, is written. The concepts for VHDL simulation are taken from Intermetrics' simulator, and from Comeau's parallel VHDL simulator called PVSIM. The parallelization of VSIM is accomplished with minimal changes to the application by utilizing SPECTRUM.

The parallel simulation protocol is implemented using SPECTRUM "filters." This provides a level of modularity that aids future experimentation with new protocols and instrumentation.

Various circuits are implemented and tested. Also, feedback among LPs is demonstrated.

1.6 Limitations.

1.6.1 VHDL Source Code Limitations for VSIM. The subset of circuits that can be simulated with VSIM includes structural descriptions of logic gates and other simple processes. Circuits are created the same way as for Intermetrics' circuits, with the following limitations:

Signals can be bits or bit-vectors; however, bit-vector inputs must be described one bit at a time, e.g., `Bus(0) <= '1' after 10 ns;`

Processes should be one-line descriptions (`Out1 <= In1 AND In2 after gate_delay;`);

```
    wait on a, b, c;  
    -- process description here  
end process;
```

```
process(a,b,c)  
begin  
    -- process description here  
end process;
```

```
process  
begin  
    -- process description here  
    wait; -- that is, wait indefinitely  
end process;
```

It is uncertain how functions and procedures may act in VSIM. For example, functions to describe multi-valued logic—or bus resolution—have not been implemented or tested.

As in the case with functions, VHDL attributes, “buffer” ports, file I/O, etc., have not been implemented or tested.

1.6.2 Postprocessor Limitations. A postprocessor, `pbuild`, is designed to transform Intermetrics generated intermediate C code for parallel simulation with VSIM. Therefore, the postprocessor only works for Intermetrics-generated intermediate C code.

The postprocessor depends heavily on recognizing unique patterns in the intermediate C code. This is accomplished using `lex`, a UNIX-based lexical analyzer. If future enhancements are to be made to the postprocessor, or if the subset of VHDL circuits is to be expanded, each step of the postprocessor should be re-evaluated for possible impact.

1.6.3 *VSIM limitations.* The user must first run the parallel simulator on one node to identify behavior id's.¹ This is accomplished by enabling a "MAPPING" definition in the simulator. Only then can a circuit-to-process mapping be defined.

Circuit partitioning must be done "by hand," i.e., the user creates the appropriate files to define logical process (LP) relationships and behavior-to-LP assignments.

The "receive message" filter used with SPECTRUM is based on a current filter called "chan-clocks." However, the new filter is modified to have access to the local LP's next event time in VSIM; therefore, the *protocol* (in the SPECTRUM filter) is modified in an *application specific* manner.

When OUTPUT is defined in VSIM, *every* signal change is reported. This becomes a bottleneck in parallel simulations on Intel Hypercubes, as processors contend for common resources, e.g., the host operating system and the disk drives.

1.7 Thesis Overview.

Chapter 2 analyzes the current research efforts in parallel discrete-event digital simulation and how they relate to this thesis. Also, other efforts in parallel VHDL simulation are reviewed. Chapter 3 provides the methodology for implementation of the post-processor, the parallel VHDL simulator, and enhancements to the parallel VHDL environment. Implementation of this methodology is discussed in Chapter 4. Chapter 5 discusses the research findings and results. Finally, conclusions and recommendations for further research are included in Chapter 6.

In addition, the following appendices are included:

-

- Appendix B: *AFIT Parallel VHDL User's Guide*. Documentation on how to prepare and run VHDL descriptions in the parallel processing environment. Also, a test case is demonstrated—using an edge-triggered D flip-flop.
- Appendix C: *Subset of VHDL Source Code for Parallel Simulation*. Describes, with examples, the subset and syntax for VHDL source that can be simulated with the parallel VHDL simulator.
- Appendix D: *Design of the Wallace Tree Multiplier*.
- Appendix E: *Summary of Performance Data*.

1.8 Summary.

VHDL models are executed sequentially in current commercial simulators. As chip designs grow larger and more complex, simulations must run faster. One approach to increasing simulation speed is through parallel processing. This research transforms the hierarchical structural models created by Intermetrics' sequential VHDL simulator into models for parallel execution on the Intel iPSC/2 and iPSC/860 Hypercubes.

II. Background

2.1 Overview.

In this chapter, several simulation techniques are discussed, including traditional simulation techniques on sequential machines, distributed simulation techniques, and digital logic simulation. Also, previous attempts to parallelize VHDL are reviewed.

2.2 Traditional Simulation.

Many real-world systems can be modeled and simulated, using computers, to study their behavior under various conditions. Examples of simulators include battlefield simulators, flight simulators, simulations of factory assembly lines, electronic circuit simulations, etc. In *continuous simulations*, the state of the model may change continuously over time. *Discrete-event simulations* are used to model processes whose states change discretely at specified points in time, as shown in Figure 1 (27). Continuous systems, like digital circuits, may also be modeled with discrete-event simulations.

Sequential simulators usually utilize three data structures (15):

1. The *state variables* which describe the state of the system.
2. An *event list* which contains the schedule of all future events.
3. A *global clock* variable to maintain the simulation time.

There are two main methods of implementing discrete simulations—*time-driven* simulations and *event-driven* simulations. In time-driven simulations, the global simulation clock is used to advance the simulation uniformly through time. With respect to digital circuits, the time-driven approach is not very efficient. If a circuit is in a quiescent state for a long period of time, waiting for the clock to advance becomes time consuming and

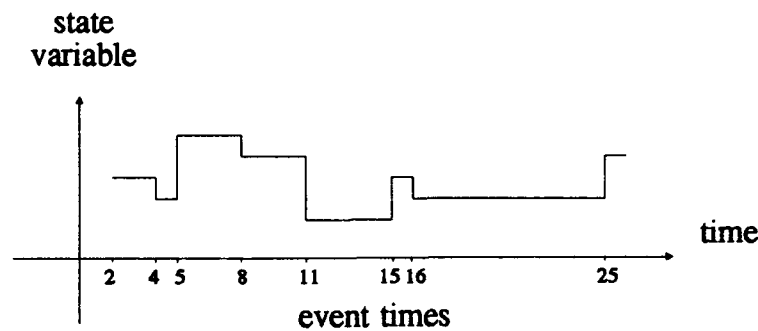


Figure 1. Response Measurement from a Discrete-Event Simulator (27)

processes *schedule* their outputs on a global event list. Then the simulation clock can advance from one event time to the next, since no computations need to occur between event times (26:136-137).

2.3.1 General Performance Model. The use of a global clock in distributed simulation constitutes a bottleneck because the LPs would all operate in lock-step. At any global time t , a number of LPs may have nothing to do. In *asynchronous* models, however, each LP contains a local virtual time (LVT), and the LPs are allowed to progress at irregular intervals. In most models, LPs communicate via time-stamped messages in the form of tuples, (t_k, m_k) , where m_k is the message sent at LVT t_k (7:199). The specific rules for message passing depend on the particular protocol.

A global event-list would also be a bottleneck in distributed simulation. Therefore, each LP usually maintains its own event-list, or queue. Events either received or self-generated can be scheduled in the local event-list, if necessary, as well as sent to "downstream" LPs, as required by the model (7:198).

2.3.2 Speed-Up and Efficiency of Distributed Simulations. If the simulation time for p processors is T_p , and the time for the same simulation on one processor is T_1 , then the *speed-up* of the distributed simulation is T_1/T_p . An ideal speed-up would be p . The *efficiency* of the simulation is therefore the speed-up divided by p . The efficiency indicates how much the communications overhead, time-management, amount of concurrency, and load imbalance among LPs deters the overall speed-up (29:43) (14).

2.3.3 Distributed Simulation Protocols. Asynchronous simulation protocols can be loosely classified as either *conservative* or *optimistic*. Conservative protocols allow an LP to advance its LVT only when it is absolutely certain it cannot receive an event with a time-stamp less than the new LVT. Optim

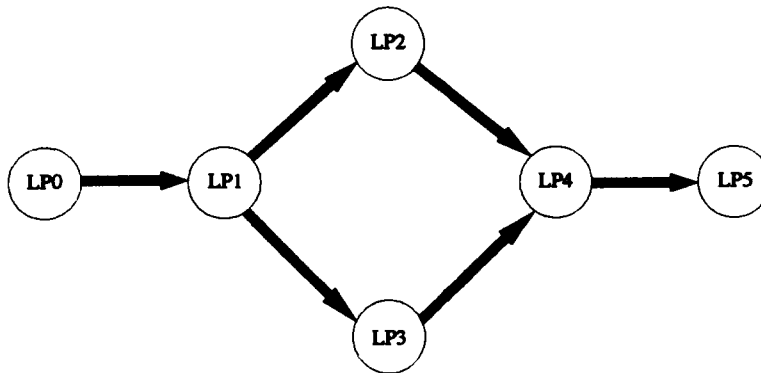


Figure 2. A Distributed System That Does Not Progress (25:56)

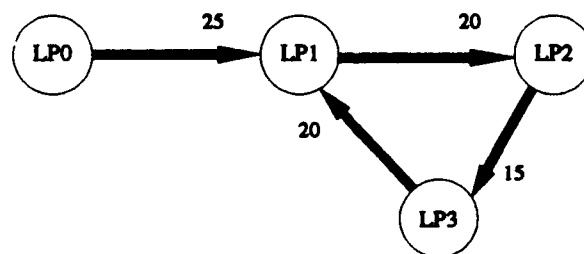


Figure 3. A Distributed System That Deadlocks (25:56)

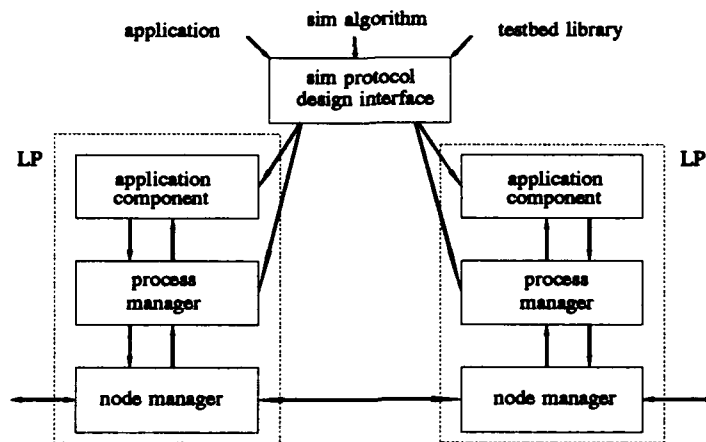


Figure 4. Block Diagram of the SPECTRUM Testbed (31)

using Reusable Modules) (32). SPECTRUM is

2.5 Other Parallel VHDL Research.

In 1989, Proicou (28) developed a distributed system consisting of a scalable kernel that supports VHDL simulations on the Intel iPSC/2 Hypercube. The distributed simulation kernel was an extension of

eight processors exhibited a speedup at least twice that of simulations on one node. His results led him to the following conclusions:

- Minimize and balance the number of active signals in a logical process.
- Carefully modify the Inter

2.6 Summary.

Discrete-event simulation mechanisms are commonly used in sequential simulations. By introducing the concept of *logical processes*, *local virtual time*, and *message-passing*, asynchronous simulation protocols can extend simulation principles to exploit parallel and distributed computers. The *conservative* Chandy-Misra protocol guarantees an LP does not receive messages out of order with respect to time, but a mechanism must be provided to avoid or detect deadlock. The *optimistic* method of Time Warp allows LPs to proceed at their own pace based on present information. If a message comes in with a time stamp in the past, then an LP must *roll back* to that simulation time in order to handle the message.

As interest in increasing the performance of VHDL grows, a number of research efforts have been conducted to investigate ways to map VHDL simulations to parallel processors. This thesis continues the work initiated by Comeau—mapping Intermetrics' VHDL capabilities to the Intel iPSC/2, and now also to the iPSC/860.

III. Methodology

3.1 Introduction.

When a VHDL circuit is compiled

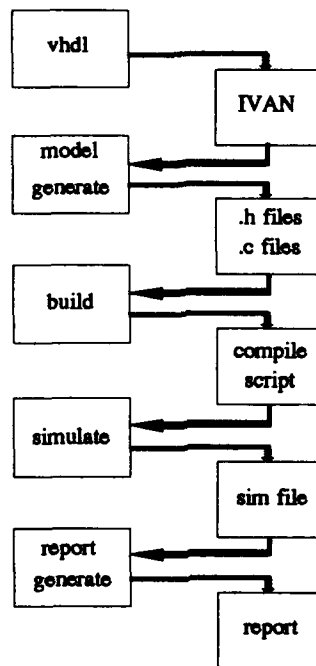


Figure 5. Simulation Session Using Intermetrics' Toolset (10:3-8)

to the designer; however, for parallel simulations, they are transformed into files that are compatible with VSIM.

In the *build* phase, a compilation script is generated that compiles and links the C modules with Intermetrics' simulator modules for operation. Now, the circuit can be simulated with the *sim* command, and a report can be generated with the *rpt* command using Intermetrics' report control language.

For parallel operation, the intermediate C code is transformed into C code that can be linked with VSIM and run on the hypercube, as shown in Figure 6. For this to happen, the code is transformed using a postprocessor

```

typedef struct BHINSTS {           /* behavior instance */
    BHKIND prty;                  /* kind (user, system, etc.) */
    INT32 id;                      /* id */
    ERRRT (*exec)();              /* behavior function */
} BHINST;

```

Figure 7. Basic Structure for Behavior Instances

```

typedef struct {                  /* signal record */
    UINT32 id;                    /* id */
    char *name;                   /* name */
    unsigned size: 4;             /* size of data value (bytes) */
    UINT32 cval;                  /* current value (offset) */
    CONNT *conns;                 /* behavioral connections */
} SRREC;

```

Figure 8. Basic Structure for Signal Records

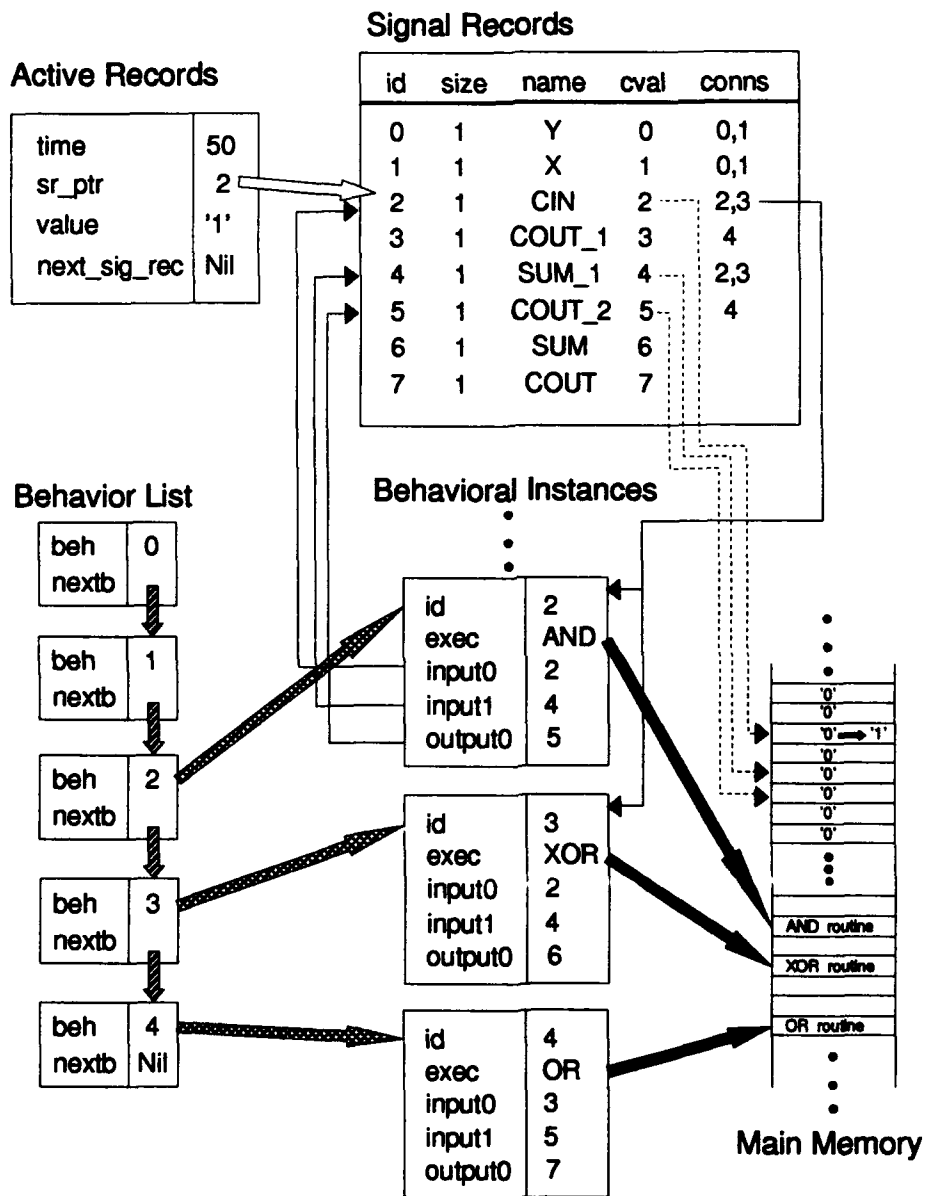
identify each signal's connections). The current value field is an offset from a global address space whose base is denoted by the global variable `cv`. See Figure 8 for an example of the signal record structure.

- **Behavior List.** This list contains all behaviors scheduled to execute for the current simulation time. At the beginning of the simulation ($t = 0$), all behaviors are scheduled for execution to initialize their input and output values. As behaviors are executed, they are removed from the list. After the simulation clock advances past zero, signal changes cause affected behaviors to be re-scheduled and re-executed. The behavior list is a simple linked-list called `tmpbeh`, see Figure 9.²

```

typedef struct TMPKS {          /* behavior list */
    BHINST *beh;               /* behavior instance pointer */
    struct TMPKS *nextb;       /* next behavior */
} TMPK;

```



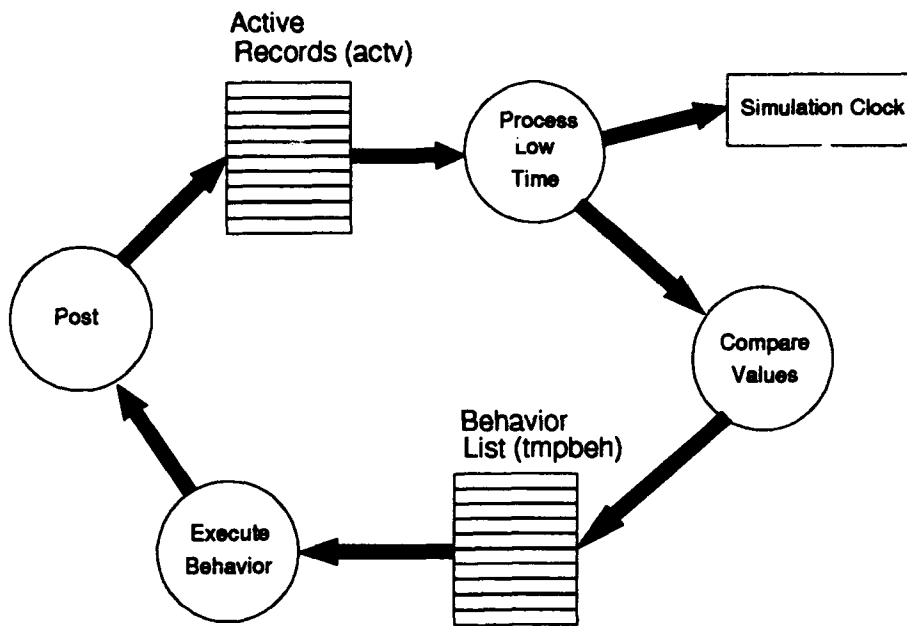


Figure 12. The VHDL Simulation Cycle (10:3-15)

the simulation:

- **post.** Posts each event to the active record list whenever a behavior has executed.
- **get_low_time.** Returns the lowest next-event time from the active records list. The simulation clock is updated to this “low time.” Records with this time are removed from the active record list and sent to the **compare_values** routine.
- **compare_values.** Compares the new data value of each event (new to the old data value in memory that is associated with that event’s behavior instance, i.e., *circuit component*). If the value is the same, the event is simply ignored (the message

```

sim_it ()
{
    SIG_REC *signal;

    /* while active record list and behavior list are not empty */
    while (actv != NULL || tmpbeh != NULL) {
        while (tmpbeh != NULL) {
            execute_behavior();          /* execute behavior and post */
            remove_behavior();
        }
        update_sim_time(get_low_time()); /* process low time */
        while (signal = active_exists(*sim_time)) {
            if (unchanged(signal)) {     /* compare values */
                remove_signal(signal);
            }
            else {
                update_signal(signal);
                schedule_behaviors(signal);
                remove_signal(signal);
            }
        }
    }
}

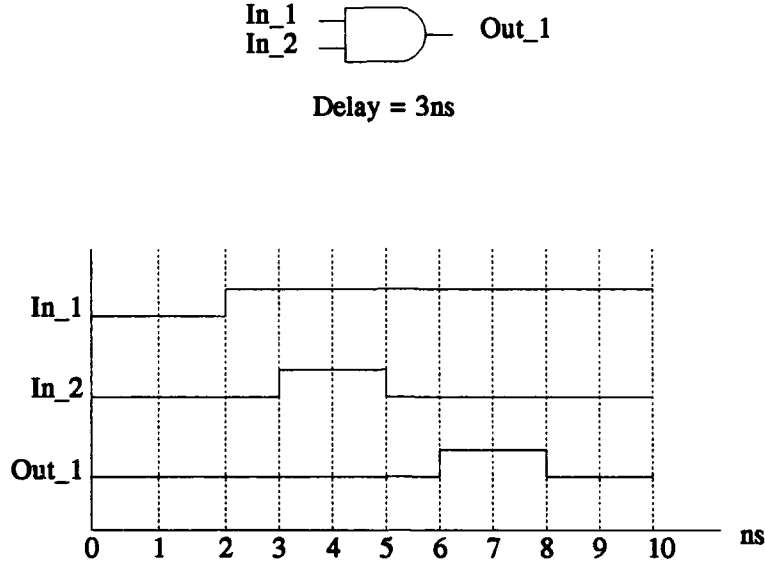
```

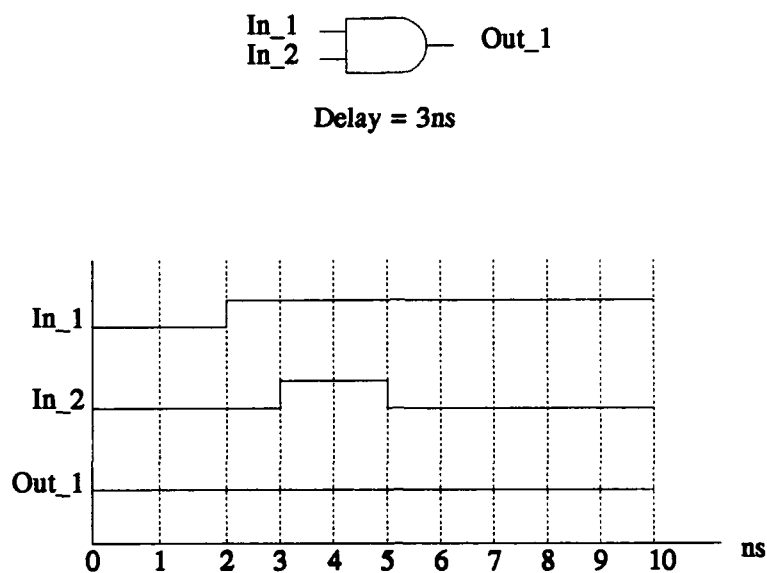
Figure 13. Main Simulation Loop in VSIM

At the beginning of the simulation, input signals are present in the active record list, and all behaviors are scheduled for execution at $t = 0$. The simulation starts at `execute_behavior`. The main (sequential) simulation loop in VSIM is shown in Figure 13. This Figure shows that the simulation cycles from executing behaviors to extracting signal changes until the active list and behavior list are empty. Specifically, while either list is not empty, perform the following:

1. Execute all behaviors on the behavior list, posting the resulting signals after each execution.
2. Update the simulation clock to the next lowest time on the active list.
3. Extract every active record with a time-tag equal to the simulation clock.

4. If the active records indicate a signal change (when compared to their current value in memory), then update the signal's value in memory and schedule affected behaviors.
5. Go back to step 1.





- Remove calls to trace routines and other trace statements. VSIM does not support tracing capabilities.
- Modify the `mksig()` function call to include a field for the signal name. This is so VSIM output can refer to signals by name instead of identifier.
- Modify the behavior functions to report the name of the entity

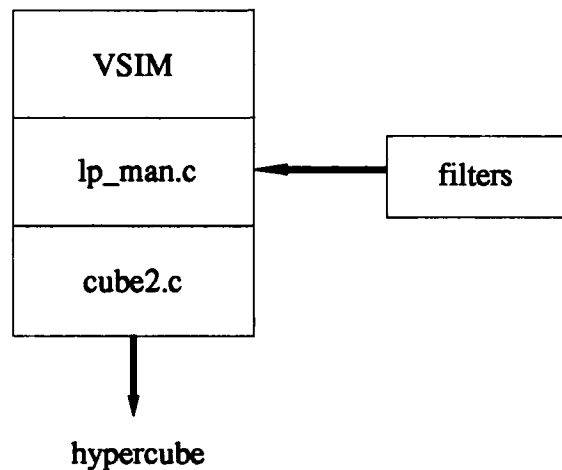


Figure 16. VSIM on the SPECTRUM Testbed (One LP Shown)

The hardware interface to the Hypercubes is provided in the functions in `cube2.c`. In general, `lp_man.c` makes these calls, and the application (VSIM) makes only LP-level calls. LPs can be partitioned among processors in a number of ways. Because of the multitasking capabilities of the Intel 80386, a “logical process” does not have to correspond to a “physical processor.” Therefore, a simulation with eight LPs can be partitioned among one to eight processors of the iPSC/2.⁷ On the iPSC/860 Hypercube, however, there must be a *one-to-one* mapping of LPs to processors, because each i860 processor does not support multitasking.⁸

3.7.2 The SPECTRUM/VSIM Filters. The SPECTRUM filters for VS

```

typedef struct event {
    int from_lp; /* lp id of lp sending event */
    int to_lp;   /* lp id of destination lp */
    int time;    /* timestamp of event */
    int event;  
```

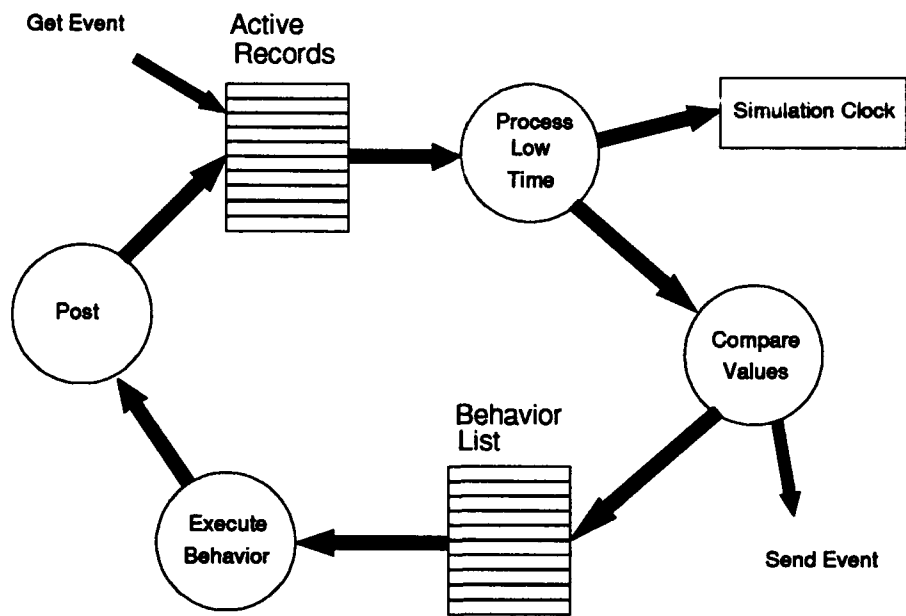



Figure 18. Parallel VHDL Simulation Cycle Shown for One LP

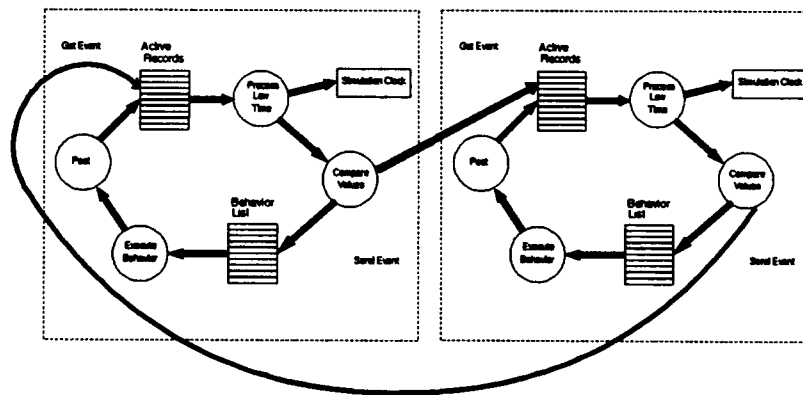
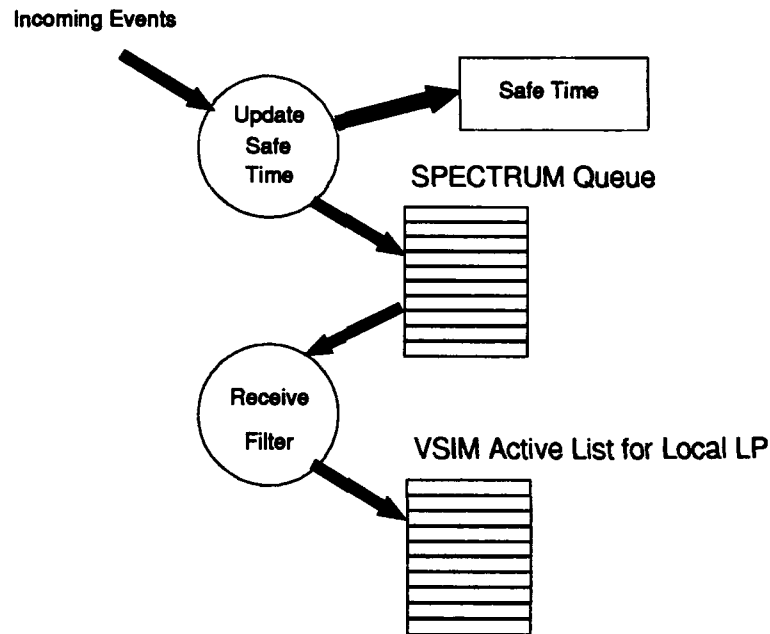


Figure 19. A 2-LP configuration



```

sim_it ()
{
    SIG_REC *signal;

    while (*sim_time < MAXTIME) {
        while (tmpbeh != NULL) {
            execute_behavior();          /* execute, and post */
            remove_behavior();
        }
        get_signal();                    /* get from other LP and post */
        update_sim_time(get_low_time()); /* process low time */
        while (signal = active_exists(*sim_time)) {
            if (unchanged(signal)) {     /* compare values */
                remove_signal(signal);
            }
            else {
                update_signal(signal);
                schedule_behaviors(signal); /* including sending to other LPs */
                remove_signal(signal);
            }
        }
    }
    end_sim();
}

```

Figure 21. Main VSIM Simulation Loop Modified for Parallel Operation

```
int lp_own[MAX_BEHAVIORS];          /* node location of each behavior */
```

Figure 22. Structure Identifying LP Ownership of Each Behavior

```

typedef struct {
    UINT32 id;           /* signal record */
    char *name;          /* id */
    unsigned size: 4;    /* name */
    UINT32 cval;         /* size of data value (bytes) */
    CONNT *conns;        /* current value (offset) */
    BOOL i_own;          /* behavioral connections */
} SRREC;               /* for LP ownership */

```

Figure 23. Basic structure for Signal Records Modified to Identify LP Ownership

is set to **TRUE** for that LP. LPs are responsible to send and/or report signal changes for those signals that they “own.”

3.8 Summary.

VHDL circuits are compiled with the Intermetrics VHDL toolset, and intermediate C code is intercepted and transformed to run with AFIT’s parallel VHDL simulator. VSIM runs either sequentially on a single processor, or in parallel on the Intel iPSC/2 or iPSC/860 Hypercubes. For parallel simulations, VSIM runs over the SPECTRUM testbed. This allows various protocols to be tested by changing filters instead of making significant modifications to VSIM. Behavioral instances are grouped into LPs and the LPs are distributed among the Hypercube’s processors.

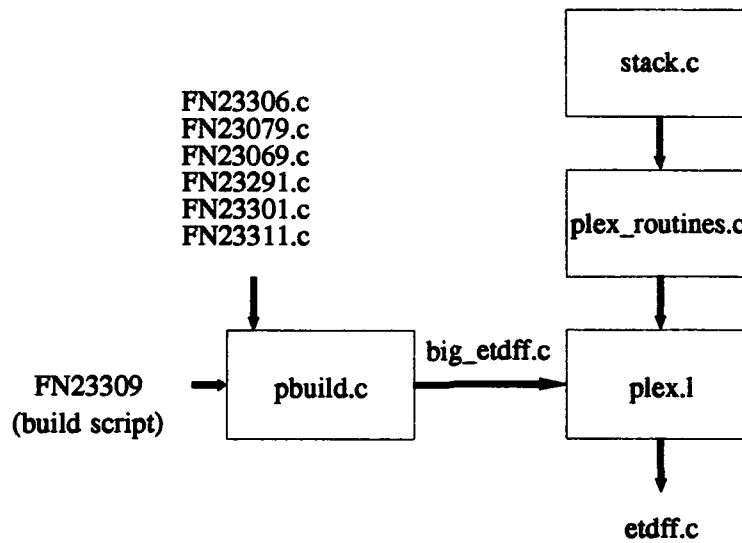
This chapter identified the key data structures, the simulation cycle, and the methodology for breaking VHDL simulations into multiple LPs and running on multiple processors.

IV. Implementation

4.1 Introduction.

```

#!/bin/csh
if ( $?VHDL_LIBSIM == 0 ) then
    if ( ! -e /usr/local/lib/libsim.a ) then
        echo NOLIB >
```



or iPSC/860 and run with VSIM. The following steps are taken to transform the intermediate C code:²

1. The intermediate C

8. The `main()` routine has six subroutine calls. The four in the middle are deleted. These functions are either not supported by the current VHDL subset, or have been replaced by `init_cv()` and `sim_it()` above.

11. Needless function calls are deleted. This was recommended—but not implemented—by Comeau since his data transformations were done by hand. Instead of deleting the calls, he wrote “dummy” functions. The following function calls are not required and are deleted:

- `close_sigdict()`
- `m_int_type()`
- `m_real_type()`
- `m_real_type()`
- `m_signal()`
- `pop()`
- `push()`
- `read_input()`
- `rmtrrec()`
- `rptstats()`
- `rpterr()`
- `Start_Nonarray_Comp()`
- `sched()`
- `timer()`
- `tpop()`

12. Every behavior instance’s “function behavior” is modified to report it’s entity/architecture name if `MAPPING` is defined in `VSIM` and the boolean variable `mapping` is still true. Each of these function declarations is of the form `Zxxxxxxx_xxxx(bi)`. Inside the function, after local declarations, put the following:

```
#ifdef MAPPING
    if(mapping)
        printf("%s\n", Zxxxxxxx_xxxx_trcbck);
#endif
```

This step is also new, and an example is shown before and after in Appendix F.

ws	<code>[\t]*</code>
comment	<code>(\\/*)[^\n]*\n</code> </

```

{comment}      { lineno++;          /* assumes comments on one line */
                numcomments++;      /* count the comments */
                ECHO;                /* echo comment
```

3. *Step 3.* The paths of all remaining include directives are modified in `check_include()`.
4. *Step 4.* `{trace...}` is changed

Approx lines:	2710
Comments:	5
#include directives modified:	5
#include directives removed:	13
{trace... changed to {... :	28
if(trceqp) tests removed:	35
"trace" or "TRAREC" lines removed:	223
Z1xxxxxx() calls removed:	4
Z5xxxxxx() functions modified:	1
Scalar "mksig" assignments modified:	18
Bit vector "mksig" assignments modified:	0
#ifdef MAPPING added:	14

Other function calls removed:

close_sigdict():	1
m_int_type():	0
m_real_type():	1
pop():	21
push():	21
read_input():	1
rmtrrec():	0
rptstats():	1

- The addresses of any filters used by all LPs.

LP relationships are specified by the user in a `lp.ar`

4.3.2 Implementation of SPECTRUM Filters for VSIM. The filters are used to implement the null-message protocol for parallel simulations. The theory behind this protocol is discussed in Chapters 2 and 3. The filters used by VSIM are based on an existing set of filters called

by `lp_get_event()`. When VSIM receives the NULL pointer, it proceeds without adding a new record to the local active list.

If the receive filter cannot return a valid message and the next event time is greater than the safe time, then the filter blocks after sending a null message to every downstream LP guaranteeing a message is not sent any sooner. In this way, deadlock is avoided. The rule, as discussed in Chapter 3, is an LP sends a null message to every downstream LP with a time stamp equal to either VSIM's next event time or the sending LP's safe time plus output delay.

4.3.3 Termination. When an LP has completed the simulation, it builds a null message with the maximum simulation time and sends it to all downstream nodes. Then it calls `node_terminate`, which signals to the host that the LP's simulation has completed. Ideally, a terminate filter should be used instead of relying on the application to create and send a null message *and* make a node-level function call.⁸

⁸Such a filter now exists in the latest version of SPECTRUM. VSIM uses this new version, but it does not use a terminate filter. Modification should be relatively straightforward and simple.

V. Results

5.1 Introduction.

TIME	-----SIGNAL NAMES-----				
(NS)	CIN				

```
10 ns, X(0) from 0 to 1
10 ns, X(2) from 0 to 1
10 ns, X(4) from 0 to 1
10 ns, X(6) from 0 to 1
10 ns, Y(2) from 0 to 1
10 ns, Y(3) from 0 to 1
10 ns, Y(6) from 0 to 1
16 ns, Z(0) from 0 to 1
16 ns, Z(3) from 0 to 1
16 ns, Z(4) from 0 to 1
21 ns, Z(3) from 1 to 0
21 ns, Z(7) from 0 to 1
30 ns, CIN from 0 to 1
30 ns, X(0) from 1 to 0
30 ns, X(1) from 0 to 1
30 ns, X(2) from 1 to 0
30 ns, X(3) from 0 to 1
30 ns, X(4) from 1 to 0
30 ns, X(5) from 0 to 1
30 ns, X(6) from 1 to 0
30 ns, X(7) from 0 to 1
30 ns, Y(0) from 0 to 1
30 ns, Y(1) from 0 to 1
30 ns, Y(2) from 1 to 0
30 ns, Y(3) from 1 to 0
30 ns, Y(4) from 0 to 1
30 ns, Y(5) from 0 to 1
30 ns, Y(6) from 1 to 0
30 ns, Y(7) from 0 to 1
30 ns, Z(5) from 0 to 1
```

Figure 32. Sample VSIM Output for Carry Lookahead Adder

6. Run the simulation in any parallel configuration, concatenate the LP output files, sort them by time and signal name, and use `diff` to compare them with the validated output.

indicate that these larger circuits can be correctly simulated; therefore, more aggressive partitioning strategies can be investigated in the future.

5.4 Explanation of Charts.

